ORACLE®

Formula Language User's Guide

Version 10g Release 3 (10.3)

Formula Language

Formula is primarily an expression language, which is tightly integrated with Designer. The tight integration allows you to specify/embed small snippets of an executable code, which typically evaluates to a value. How the evaluated value is used depends on the context in which the formula is used. For instance, if it is used for validation of a field, then the formula should evaluate to a boolean value. A return value of true would indicate that the validation succeeded.

Formula scripts are written using plain ASCII characters. The lexical translations result in sequence of input elements, which are whitespaces, comments and tokens. The token comprises identifiers, keywords, literals, operators and separators of Formula grammar. Where possible, lexical similarity with Java/C++ has been maintained.

Expression Language

Formula is primarily an expression language. Typically, you write a small formula snippet, which evaluates to a value. The returned value is used, depending on the context in which formula is used.

For example, the following snippets are valid formula
1. 10
2. 5 + 3
3. "Hello"
4. a + b
5. a == true

In some cases, you may want to perform a non-trivial computation. The computation may require loops to perform a complex calculation or local variables to hold state. In general, you may want to perform a computation, which cannot be expressed in a simple expression (pun intended). Formula language supports local variables and statements similar to languages such as Java and C++.

From its simplest form of a simple expression, the formula language scales up to support structured constructs such as **if**, **for**, **foreach**, **break**, **continue**, **return** statements. In all cases, the return value of the last *top-level* expression/statement is treated as the return value of the snippet.

```
def sum =0;
for(def i =0; i<10;++i) {
    sum += i;
}
sum;
```

In this snippet, the formula returns the value of the variable sum. You can also use an explicit **return** statement to return a value. Using a **return** statement allows you to return a value in the middle of the snippet as well as at the end as shown in the following snippet.

```
def number = 121;
for (def i = 2; i <= (number/2); i++) {
    if (number % i == 0) {
        return false;
    }
}
return true;
```

See Also:

Lexical Structures
Functions
Statements

Lexical Structures

Formula code is a collection of whitespace, identifiers, keywords, literals, comments, operators and separators.

Whitespace
Identifiers
Keywords
Literals
Comments
Operators
Separators

See Also:

Formula Language
Functions
Statements

Whitespace

Formula language is a freeform language. You do not have to indent anything to get it to work properly. A formula could be written all on one line:

```
def sum = 0; for (def i = 1; i <= 4; ++i) {sum += i;} return sum;
```

or in any other strange way you feel like typing it, as long as there is at least one space, tab, or new line between each token that is not already delineated by an operator or separator. This, for example, will produce the exact same formula as the single line above.

```
def
sum =
0;
for(def i  = 1;i  <=
4;
++i) {sum
+= i;} return
sum;
```

See Also:

[Identifiers](#)
[Keywords](#)
[Literals](#)
[Comments](#)
[Operators](#)

Identifiers

Identifiers are used to name variables, fields, and functions. An identifier may contain the letters 'a' .. 'z', 'A' .. 'Z', the digits '0' .. '9', and the characters '_' and '$'. There is no restriction on the length of the identifier name. All identifiers in Formula are case sensitive. By convention a variable starting with '$' is used for local variable.

See Also:

[Whitespace](#)
[Keywords](#)
[Literals](#)
[Comments](#)
[Operators](#)

Keywords

The list of reserved keywords is given below. These are reserved for use and cannot be used as identifiers.

for	foreach	in	if	else
switch	while	do	delegate	lambda
public	private	protected	final	native
abstract	volatile	synchronized	int	long
short	char	float	double	boolean
void	def			

There are very few keywords in list above that are actively used in the language. Many of these keywords are reserved for future use.

Note that, a function name can be same as the keyword provided you use a different case at call point. For example, to call the 'If' function you need to use If(…) and not if(….) .

See Also:

Whitespace
Identifiers
Literals
Comments
Operators

Literals

All literals available in Java are supported in Formula. The only difference is the lack of unicode support for string and character literals.

Type	Description	Examples
String	The value should be enclosed within double quotes. Supports Unicode literals (\unnnn) Supports hex literals (\#xHH) Standard escape sequences (as in Java) \r, \n, \b, \t, etc can be used in the string. **See Also:** Template String Literal	"hello", "\u0040 foo"
Character	The single character value should be enclosed with single quotes. Supports Unicode literals Supports hex literals (\#xHH) Standard escape sequences (as in Java) \r, \n, \b, \t, etc can be used	'a', '\#x13', '\n'
Integer	Supports base10 (normal), octal & hex literals	31, 067, 0x20
Long	Supports base10 (normal), octal & hex literals Should be suffixed with 'l' or 'L' to distinguish it from Integer literals	1234567L
Boolean		true, false
Float	Java/C++ style IEEE floating	23.1f, 10f

	point literals. Should be suffixed with 'f'	
Double	Java/C++ style IEEE double precision literals. Can be suffixed with 'd'	23.1, 23.1d
BigInteger	Supports base10 (normal), octal & hex literals Should be suffixed with 'n'	1234567890123456789012345n
BigDecimal	Should be suffixed with 'm'	1.1234567890123456789012345m

See Also:

Whitespace
Identifiers
Keywords
Comments
Operators
Template String Literal

Comments

All types of comments supported by Java/C++ are treated as comments. This includes the single-line comment using // syntax and the multi-line comment using /*   */ syntax.

See Also:

Whitespace
Identifiers
Keywords
Literals
Operators

Operators

An operator is something that takes one or more arguments and operates on them to produce a result.

All operators available in Java are supported in Formula with identical precedence, arity, and commutativity. Some of the operators behave slightly differently in Formula. For example, unlike in Java, assignment operator does not return the value.

So the following is not legal,

`a = b = c;`

Similarly, the ++ operator does not return the incremented value.

In general, the intention is to allow either a side effect (such as for ++) or a return value (such as for + operator) for each operator. Operators with both side effect and a return value are not available.

Supported Operators:

[Arithmetic Operators](#)
[Unary Arithmetic Operators](#)
[Relational Operators](#)
[Logical Operator](#)
[Bit operators](#)
[Assignment operators](#)
[Array Access](#)

See Also:

[Operator Precedence](#)
[Whitespace](#)
[Identifiers](#)
[Keywords](#)
[Literals](#)
[Comments](#)

Arithmetic Operators

Operator	Function Name	Comments
+	Plus	This binary operator works on all numeric types and has the usual meaning. The plus operator is *overloaded to work with Strings* like in Java). If either of the values is a String then values are concatenated. It is equivalent to calling the concat function with the two arguments. e.g. return ("Sum of " + i + " and " + j + " is: " + (i+j));
-	Minus	This binary operator works on all numeric types and has the usual meaning.

*	Multiply	This binary operator works on all numeric types and has the usual meaning.
/	Divide	This binary operator works on all numeric types and has the usual meaning.
%	Remainder	This binary operator works on all numeric types and has the usual meaning.

See Also:

[Unary Arithmetic Operators](#)
[Relational Operators](#)
[Logical Operator](#)
[Bit operators](#)
[Assignment operators](#)
[Array Access](#)
[Operator Precedence](#)

Unary Arithmetic Operators

Operator	Function Name	Comments
++	Incr	Increments the value of the variable. This is supported only for local variables. Unlike Java/C++ this does not return the incremented value. So there is no difference between pre-increment & post-increment, though you can use both styles. e.g. a) ++i //legal b) t++ //legal c) t = ++i // illegal
--	Decr	Decrements the value of the variable. This is supported only for local variables. Unlike Java/C++ this does not return the decremented value. So there is no difference between pre-decrement & post-decrement, though you can use both styles. e.g. -–i //legal

		t-- //legal
		c) t = --i // illegal
+	UnaryPlus	The sign of the operand is unchanged. Usually this is not used.
		e.g.
		+a //if the value of 'a' is –5, it continues to be –5.
-	UnaryMinus	This is equivalent of multiplying the operand by -1.
		e.g.
		-a //if the value of 'a' is 5, it becomes –5.

See Also:

Arithmetic Operators
Relational Operators
Logical Operator
Bit operators
Assignment operators
Array Access
Operator Precedence

Relational Operators

In order to compare two values, formula has the following set of relational operators that describe equality and ordering. Most types, including integers, floating point numbers, characters, booleans, dates, strings can be compared using the equality test, ==, and the inequality test, !=.

Number, string, character and date types can be compared using the ordering operators. The relational operators always return a boolean value, true or false.

Operator	Function Name	Description
<	Less	Less than
>	Greater	Greater than
<=	LessEqual	Less than or equal to
>=	GreaterEqual	Greater than or equal to
==	Equal	Equal to

!=	NotEqual	Not equal to

See Also:

Logical Operator

The Boolean logical operators summarized below operate only on boolean operands. The binary logical operators operate on a one or more boolean values to form a resultant boolean value.

Operator	Function Name	Description
&&	And	The logical 'And' operator returns true only if both the operands are true e.g. true && true //returns true true && false //returns false false && true //returns false false && false //returns false
||	Or	The logical 'Or' operator returns true if either or both of the operands are true e.g. true && true //returns true true && false //returns true false && true //returns true false && false //returns false
? :	If	The syntax of the conditional (ternary) operator is given below. <expression1> ? <expression2> : <expression3> Here, expression1 should be a Boolean expression;

		expression2 and expression3 should be of the same type. First expression1 is evaluated. If its value is true, then expression2 is evaluated and its value is returned as the result of the operator. If its value is false, then expression3 is evaluated and its value is returned as the result of the operator.
		e.g.
		(1 < 2) ? 1 : 2 //returns 1
		(1 < 2) ? "small" : "big" //returns "small"
!	Not	The logical 'Not' operator returns the complement of the input operand.
		e.g.
		!true //returns false
		!false //returns true

See Also:

[Arithmetic Operators](#)
[Unary Arithmetic Operators](#)
[Relational Operators](#)
[Bit operators](#)
[Assignment operators](#)
[Array Access](#)
[Operator Precedence](#)

Bit Operators

Operator	Function Name	Description
<<	LeftShift	The syntax of the left shift operator is given below: value << num It shifts all the bits of the value operand to the left by the number of times as specified by the num operand. It should be noted that the value operand should be of either integer or long type and the num operand should be of integer type. e.g. 16 << 2 // returns 64
>>	RightShift	The syntax of the right shift operator is given below: value >> num It shifts all the bits of the value operand to the right by the number of times as specified by the num operand. It preserves the sign as it extends the sign bit. It should be noted that the value operand should be of either integer or long type and the num operand should be of integer type. e.g. 64 >> 2 // returns 16 -64 >> 2 // returns -16
>>>	UnsignedRightShift	The syntax of the unsigned right shift operator is given below: value >>> num It shifts all the bits of the value operand to the right by the number of times as specified by the num operand. It does not preserve as it extends with zero bit. It should be noted that the value operand should be of either integer or long type and the num operand should be of integer type. e.g. 64 >>> 2 // returns 16

		-64 >>> 2 // returns -1073741808
&	BitAnd	The Bitwise 'And' operator produces a 1 bit if the corresponding bits in the operands are 1. A zero is produced in all other cases. e.g. 42 & 15 //returns 10
|	BitOr	The Bitwise 'Or' operator combines bits such that, if any one of the corresponding bits in the operands is a 1, then the resultant bit is a 1. e.g. 42 | 15 //returns 47
^	BitXOR	The Bitwise 'XOR' operator combines bits such that, if exactly one of the corresponding bits is 1, then the resultant bit is 1. Otherwise, the resultant bit is zero. e.g. 42 ^ 15 //returns 37
~	BitNot	Not supported

See Also:

Arithmetic Operators
Unary Arithmetic Operators
Relational Operators
Logical Operator
Assignment operators
Array Access
Operator Precedence

Assignment Operators

Operator	Function Name	Description
=	Set	The RHS and LHS should be type compatible. Unlike Java/C++ this operator does not return the assigned value. So a = b = c style of assignment is illegal.
+=	PlusEqual	Adds the RHS value to the LHS variable Only works on local variables (LHS). Does not return the assigned value.
-=	MinusEqual	Subtracts the RHS value from the LHS variable Only works on local variables (LHS). Does not return the assigned value.
*=	MultiplyEqual	Only works on local variables (LHS). Does not return the assigned value.
/=	DivideEqual	Only works on local variables (LHS). Does not return the assigned value.
%=	ModulaEqual	Only works on local variables. Does not return the assigned value.

See Also:

Arithmetic Operators
Unary Arithmetic Operators
Relational Operators
Logical Operator
Bit operators
Array Access
Operator Precedence

Array Access

Operator	Function Name	Description
[]	At	Access the element from a collection/array/aggregate type. The index of the element must be specified within the square brackets. The index is zero based, that is to get the first element you need to use the index 0.
		This operator is available for Section, Array, String and Binary.
		A type which supports array access operator (At function) and the length() function can be used in a foreach loop.
		e.g. str[2] returns the 3rd character in the string str.
		Since most of the types in formula language are immutable, this operator can only be used to access the value at the specified index and not to assign to it. The following is illegal.
		str[2] = 'c'; // illegal

See Also:

Arithmetic Operators
Unary Arithmetic Operators
Relational Operators
Logical Operator
Bit operators
Assignment operators
Operator Precedence

Operator Precedence

In an expression there is a certain order, or precedence of operations. The following table shows the order of all of the possible Formula operations, from highest precedence to lowest:

Highest			
()	[]	.	
++	--	!	
*	/	%	
+	-		
>>	>>>	<<	
>	>=	<	<=
==	!=		
&			
^			
|			
&&			
||			
?:			
=	op=		
Lowest			

Explicit Precedence

Since parentheses are the highest precedence, you can always throw in a few extra pair if you are not sure about implicit precedence rules or want to make sure your code is readable.

If for example, you are not sure what the following expression really means
a | 4 + c >> b & 7 || b > a % 3
Try to put in a few clarifying parentheses, such as:
(a | (((4 + c) >> b) & 7)) || (b > (a % 3))

Parentheses also allow you to explicitly set the precedence of an operation, which is often needed when combining shifting and addition like this:
a >> b + 3
Which of the following:
a >> (b + 3)
or
(a >> b) + 3
does the first expression resolve to? Since + is higher precedence than >>, you will get the first result. If you intended the second result, the parentheses are required to explicitly set the precedence of operations.

See Also:

Operators

Separators

Separators define the shape and function of your code. The table given below will define the separators and explain their use.

Symbol	Name	What it is used for
()	parentheses	Used to contain lists of parameters in function invocation and also used for defining precedence in expressions
{}	braces	Used to define a block of code and local scopes
[]	brackets	Used to declare array types (lists), also used when dereferencing array values (items of a list)
;	semi-colon	Separates statements

,	comma	Used to separate items in a array declaration
.	period	Used to separate a nested field from its parent section

See Also:

Whitespace
Identifiers
Keywords
Literals
Comments
Operators

Functions

Function names in Formula language are case insensitive. By convention, we use camel casing. The first letter also starts with a capital letter by convention.
As of this version, there is no way to define a function in the formula language. The functions are implemented natively (written in Java/C++ etc) and made available to language depending on the context in which it is used.
To invoke a function use the normal function call syntax as in Java
E.g. Left(str, 10)

The object oriented syntax is also supported. In this case the first argument of the function is pushed out followed by a '.' and the function name. In the argument list you need to remove the first argument.
The following are equivalent,

1. Left(str, 10)
2. str.Left(10)

If a function or an operator returns a value it is illegal to ignore it. For example the statement Left(str, 10); is illegal. You need to use the return value, by assigning it to a variable or by some other means. This rule is relaxed if the statement is the last top-level statement, in which case it is implicitly returned to the caller (hence the return value is not ignored).

Many functions are overloaded to work on multiple types (e.g. toText(int), toText(double)).
For some functions trailing parameters are optional. In general when a function is overloaded the semantics of the overloaded functions is kept the same.
Some functions like 'In', 'Concat' take variable number of arguments.
Like in Java/C++ you can nest function calls as given below.
Left(Right(str, 10), 5)

See Also:

[Formula Language](#)
[Lexical Structures](#)
[Statements](#)

Statements

As mentioned earlier, though formula is primarily an expression language, it nicely scales up to support statements.

Statements supported:

[Block Statements](#)
[Condition Statements](#)
[Iteration Constructs](#)
[Return statement](#)
[List Literal](#)

See Also:

[Local Variable Declaration](#)
[Variable Scope](#)

Local Variable Declaration

In any nontrivial formula, we need variables that keep track of program state. Local variables can be used to hold the state across statements.
To define a local variable use the **def** keyword.
def varName = expr;
The following snippet prints the sum of two variables.

```
def i = 123;

def j = 6456;

return ("Sum of " + i + " and " + j + " is: " + (i+j));
```

You don't have to (as a matter of fact, there is no way to) specify the type of the variable. Type is inferred from the initialization expression. It is incorrect to define a variable without initialization. Once a variable is defined with an inferred type, it is illegal to assign to it an incompatible value. For instance the following is illegal.

```
def i =10;

i = "hello" // illegal i is of type int.
```

The formula language is statically typed. Though it looks a scripting language it shares more in common with statically typed languages such as Java C++.

It should also be noted that a local variable could not hold a null value. This is especially true when a field is assigned to a local variable. It is the user's responsibility to check the field for null value (using IsNull() or IsNotNull() function) and assign only the non-null value to a local variable.

There is also another way to define a local variable. Variables which start with a $ symbol are treated as local variables. If variable with that name is not yet defined, a new variable is defined implicitly. As in the case of 'def', the type of the variable is inferred based on the initialization expression. The following snippet defines a local variable named $i of type int and initializes it to value 10.
$j = 10;

Variables such as i, $j that are defined within a formula snippet are local to that formula; they cannot be accessed from outside, say from another formula snippet.

Normal scoping rules, as in Java, apply in formula. It is illegal to define a local variable when another variable with the same name is available in the scope (either declared in that scope or inherited from the enclosing scope).

See Also:

[Block Statements](#)
[Variable Scope](#)
[Condition Statements](#)
[Iteration Constructs](#)
[Return statement](#)
[List Literal](#)

Block Statements

A group of statements can be specified with in a block. A block start with brace '{' and is terminated with brace with a group of statements in between.
The statements in the block are executed in a sequence, from top to bottom, in the lexical order. A block statement can be used in any place where a statement is allowed.

```
{
    def i = 10;
    sum = a * i;
}
```

The block statement also starts a new scope (for local variables). Variables defined in this scope are not visible outside the block.

See Also:

Local Variable Declaration
Variable Scope
Condition Statements
Iteration Constructs
Return statement
List Literal

Variable Scope

Formula local variables are valid only from the point where they are declared till the end of the block. The blocks can be nested, and each one can contain its own set of local variable declarations. You cannot, however, declare a variable to have the same name as one in an outer scope. Here is an example that tries to declare two separate variables with the same name. In Formula, this is illegal.

```
def sum = 0;
def x = 1;
{
    def y = 2;
    {
        sum += y;
        {
            def x = 3;  //illegal, variable with name 'x' already defined
            sum += x;
        }
    }
}
sum += x;
return sum;
```

See Also:

Local Variable Declaration

Condition Statements

Formula supports both "if…then…else" statement and the ternary operator. While the former allows you to use multiple statements, the latter allows only an expression.

```
if(boolExpr)
    statement
else {optional}
```

```
statement
```

If the Boolean expression evaluates to true, the 'then' block/statement is executed. If it evaluates to false, the else block, if available, will be executed. If needed, you can use block statements in the then and else clause since a block statement is also a normal statement.

The following code snippet returns the smaller between x and y. It does not use the else block.

```
def x = 10;
def y = 20;
if(x < y) {
   return x;
}
return y;
```

The following code snippet also returns the smaller between x and y. But it uses the **else** block.

```
def x = 20;
def y = 10;
if(x < y) {
   return x;
}
else {
    return y;
}
```

See Also:

Block Statements
Variable Scope
Iteration Constructs
Return statement
List Literal

Iteration Constructs

Formula supports for, foreach statements. These loop constructs support C++-style continue and break statements to alter loop execution. There is no labeled break and continue (available in Java).

Iteration statements:

for statement
foreach statement

See Also:

for Statement

Here is the syntax of the for statement:

```
for (initialization; condition; iteration) <body>
```

The **for** loop operates as follows. When the loop first starts, the **initialization** portion of the loop is executed. Generally this is an expression that sets the value of the loop control variable. It is important to understand that the initialization expression is executed only once. Next, **condition** is evaluated. This must be a Boolean expression. It usually tests the loop control variable against a target value. If this expression is true, then the body of the loop is executed. If it is false, the loop terminates. Next, the **iteration** portion of the loop is executed. This is usually an expression that increments or decrements the loop control variable.

The following code snippet returns the sum of values from 1 to 4.

```
def sum = 0;
for(def i = 1; i <= 4; ++i)
    sum += i;
return sum;
```

See Also:

foreach Statement

The 'foreach' construct is a simplified 'for' which is convenient for traversing collections and sections. Unlike the 'for' loop, you don't need to specify an initializer, condition and a increment part. The foreach, works on a collection and loops once for each element of the collection. The element of the collection is made available inside

the loop.

You can break out the loop using the break statement or skip an element using he continue statement. But there is no way to get the index of the current element in the loop.

The following code snippet returns the double value 20.34.

```
def sum =0.0;
foreach($el in [1.0, 12.34, 7] )
    sum += $el;
return sum;
```

Additionally 'foreach' allows you to specify a where clause. This is a Boolean expression, which is evaluated for every iteration, and the body of loop is executed only if it is true.

Note:

If the 'where' clause is false, it does not break out of the loop, it continues to the next iteration.

If no where clause is specified, the body is executed for all iterations.

Example

```
def colors = ["red", "green", "blue"];
foreach(def color in colors where color != "green") {
    // do something with color
}
```

Here, the body of the loop executes for all colors except "green".

The foreach construct also supports multiple nested iterations by allowing you to specify multiple loop variables and collections. A comma separates each loop specification, and the second and subsequent loops are executed as nested loops.

In the following snippet, two loops are defined using a single foreach construct. The loop on favColors is executed as an inner loop. Each loop specification can have its own 'where' clause which can refer to the current and preceding loop variables.

```
def colors = ["red", "green", "blue", "yellow", "pink"];
def favColors = ["yellow", "violet"];
foreach(def color in colors,
        def favColor in favColors where favColor == color) {
```

```
      // intersection of color and favColor

}
```

This kind of multiple loops is particularly useful when you want to loop over a deeply nested section. For instance, if you have a top-level section A with child B which itself has a child section C, to iterate over all instances of C, you can use the following snippet.

```
foreach(def A in secA,
        def B in A.B,
        def C in B.C) {

}
```

You can add where clause to each of the loop specifications to iterate over only those elements that satisfy the condition.

The foreach syntax is:

```
foreach ( <loopvar> in <collection exp> [where condition], … ) <body>
```

Everytime through the loop, the next element of the collection is bound to <loopvar>. The collection expression is evaluated only once.

See Also:

[for statement](#)
[break statement](#)
[continue statement](#)

while Statement

The **while** loop repeats a statement or block while its controlling condition is true. Here is its general form:

```
while (condition) {
   //body of loop
}
```

The condition can be any Boolean expression. The body of the loop will be executed as long as the conditional expression is true. When condition becomes false, control passes to the next line of code immediately following the loop. The curly braces are unnecessary if only a single statement is being repeated.

The following code snippet returns the sum of values from 1 to 4.

```
def sum = 0;
def i = 1;
while(i <= 4) {
    sum += i;
    ++i;
}
return sum;
```

See Also:

for statement
do-while Statement
break statement
continue statement

do-while Statement

If the conditional expression controlling a **while** loop is initially false, then the body of the loop will not be executed at all. However, sometimes it is desirable to execute the body of a **while** loop at least once, even if the conditional expression is false to begin with. In other words, there are times when you would like to test the termination expression at the end of the loop rather than at the beginning. The **do-while** loop always executes its body at least once, because its conditional expression is at the bottom of the loop. Its general form is

```
do {
    //body of loop
} while (condition);
```

Each iteration of the **do-while** loop first executes the body of the loop and then evaluates the conditional expression. If this expression is true, the loop will repeat. Otherwise, the loop terminates. As with all other loops, the condition must be a Boolean expression.

The following code snippet returns the sum of values from 1 to 4.

```
def i = 1;
def sum = 0;
do {
    sum += i;
    ++i;
}
while(i <= 4);
return sum;
```

See Also:

for statement
while Statement
break statement
continue statement

break Statement

The **break** statement is used to prematurely break out of a loop without executing the loop's body again. It should be used inside a loop construct (for & foreach) and the control jumps to the statement next to the loop.
The following code snippet checks if the number 121 is a prime or not. The **break** statement transfers control out of the **for** loop as soon as the number is found divisible (i.e. the number is not prime).

```
def number = 121;
def prime = true;
for (def i = 2; i <= (number/2); i++) {
    if (number % i == 0) {
        prime = false;
        break;
    }
}
return prime;
```

See Also:

for statement
foreach statement
while Statement
do-while Statement
continue statement

continue Statement

The continue statement is used inside a loop to skip processing the remainder of the code for this particular iteration. The execution continues with the next iteration of the loop.

The following code snippet finds the sum of all the odd numbers between 1 and 10 inclusive. Here, the **for** loop generates numbers from 1 to 10. In case of even numbers, the **continue** statement transfers control to the next iteration of the loop without incrementing the sum.

```
def sum = 0;
for (def i = 1; i <= 10; ++i) {
     if (i % 2 == 0) {
          continue;
     }
     sum += i;
}
return sum;
```

See Also:

Return Statement

Return statement can be used to cause the execution to branch back/return to the caller of the formula. The formula snippet stops executing and can optionally return a value to the caller.

As mentioned earlier, the last top-level statement/expression in the formula code is implicitly treated as the return value. You can also use a explicit return statement at the end of the code. The return statement can also be used in the middle of the code (typically conditionally) to return a value to the caller.
The following are equivalent
a) return a + b;
b) a+b

A more useful case is to return a value from the middle of the formula code.

```
if(a>b)
  return a;
else
  return b;
```

In case of multiple return statements all of them should return a value of same type (or return nothing). It is also possible to return the last expression implicitly and return another value from some where in the middle. It is necessary for the return values to be of same type.

Note that the caller of the formula depends on the context in which it is used. If it is used for enforcing a validation, its return value is used by the validator. The return value of the formula must be of a type, which is acceptable to the caller. For

instance, in case of a validation, the formula should return a Boolean value. In case of mapping, the return type should match that of the destination field.

See Also:

Block Statements
Condition Statements
Iteration Constructs
List Literal

List Literal

A list is a group of like-typed values that are referred to by a common name. The syntax given below can be used for creating lists (arrays) and it can be used with all data types.
[value1, value2, …]
For example, the following statement shows the declaration of a variable **values**, which is an array of integer values.

```
def values = [10, 30, 40];
```

The items of a list can be accessed using index. The following code snippet returns the sum of the items in the list **values**.

```
def values = [10, 30, 40];
def sum = 0;
for (def i = 0; i < Length(values); ++i) {
    sum += values[i];
}
return sum;
```

The foreach construct can also be used to iterate through the items of the list as can be seen in the following code snippet that calculates the sum of the items in the list values.

```
def values = [10, 30, 40];
def sum = 0;
foreach($val in values) {
    sum += $val;
}
return sum;
```

See Also:

Block Statements
Condition Statements

Template String Literal

One of the common requirements is generating semi-structured output messages based on a template. You can use Universal plug-in to generate such output. Though using Universal plug-in works, it is not easy to maintain it. A small change in the output structure may require substantial changes in the message definition. More importantly, you don't see how the output is generated. It is all hidden in Universal message definition as tags and delimiters.

An ideal solution is to have a template like approach supported directly in the Cartridge. Many scripting languages like Groovy and Ruby support it (to some extent). The formula language now supports this approach and it is similar to what Groovy does and many other languages are seriously considering.

Template strings are supported using a special kind of string literal. These are the differences between this special literal and a normal string literal.

1. Unlike a normal [literal](#) this starts with `@"` and ends with `"@`.

2. The literal can span multiple lines.

3. Text inside the literal is not escaped. That is, it does not support escape sequences like \n, \t, etc. If you want a linefeed, simply enter the text in the next line. If you want to enter a quote, just enter the quote character and there is no need to escape it. In most cases you should be able to enter multi line freeform text (like in a text document) and enclose it within this special quote.

See Also:
[Expressions](#)
[Scriptlets](#)
[Line Feed and Whitespaces](#)
[Directives](#)
[Summary of Template String Syntax](#)
[Error Highlighting](#)
[Syntax Highlighting](#)
[Usage Scenario](#)
[Comparison with Universal Plug-in](#)

Expressions

Support for such special string literals are even found in mainstream languages like C#. But scripting languages go one step further and allow you to embed [expressions](#) within the literal. You can do this in formula's string literals.

```
def str =  @"Reference: ${obj.ref}"@;
```

The expression is enclosed with ${ }. The value of the expression is inserted at the location where it is used. The expression needs to be valid in the context it is used. It can make use of local variables/message variables that are accessible in the formula. The expression need not evaluate to a string type. If it evaluates to a non-string type, the default text representation of the value is used. The above string literal translates to the following:

```
def str =  "Reference: " + obj.ref;
```

See Also:

Scriptlets
Directives
Template String Literal

Scriptlets

The syntax of embedding expressions in literal, works well for common cases. But when it is used for generating full-fledged messages, it breaks down because,

1. In many cases, the parts of the output should be conditionally generated. For example, in the above example, only if 'ref' has a non-null value.

2. In case of messages, certain areas of the output are recurring (repeating). In the above case if 'ref' is part of a repeating section, we may have to generate the output line once for each occurrence.

Both of the above problems can be addressed by breaking the output template into multiple parts and concatenating these parts in the formula.

```
def output = "";
if(isNotNull(obj.ref)) {
  output +=  @"Reference: ${obj.ref}"@;
}
```

The whole point of template is now lost because the formula code now dominates instead of the template. To know what is happening, you need to go through, multiple lines of literals embedded within normal formula code. Having said that, it is impossible to avoid some kind of 'if' statement since we want part of the output to be generated conditionally. The standard solution for this problem is to allow statements to be embedded in the template instead of other way around. This would keep the focus on the template, but it would still let you control the output the way

you want. This is the approach followed in JSP (which allows embedded Java statements).

In tune with other solutions, the special string literal supports embedded formula statements. The statements should be enclosed with in <% and %> markers (same as in JSP). Within the special string literal, anything enclosed within these markers will be treated as formula statements and are emitted as it is.

```
def str =  @"
<% if ( !isNull(obj.ref)) { %>
Reference: ${obj.ref}
<%} %>
"@;
```

This is almost identical to what JSP does. The above translates to

```
def str =  "";
if (!isNull(obj.ref)) {
    str += "Reference: " + obj.ref;
}
```

Though the difference is not substantial in this case, it really shines when the literal text dominates in the output string; for instance, when you generate HTML or other verbose outputs.

Just like the 'if' statement shown above you can use any formula statements; you can define variables, use a 'for/foreach' loop, etc. Just make sure that the blocks are properly closed.

Note:

Unlike a full-fledged template language like JSP, template strings are available in formula as literals. That is, the entire text of the formula code is not treated as template; only those parts that are enclosed within the special quotes (@"). This allows you to use formula statements and expressions not just inside the template string but outside as well. The template string itself is just an entity within the formula code. For instance, you can declare local variables in the formula code, which will contain some computed value. These local variables can be referenced within the template string.

```
def  totalPrice = secSumDouble(obj, "price");
def outStr = @" The total price is ${totalPrice}"@;
```

The above is equivalent to

```
def outStr = @" The total price is ${ secSumDouble(obj, "price")}"@;
```

Some would prefer the former because it is clearer.

The point is that, since the template is just a part of the formula code, you can chose to do your computations outside the template literal, thus keeping it clean.
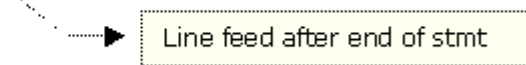
See Also:

Expressions
Directives
Template String Literal

Line Feed and Whitespaces

Linefeeds and whitespaces in the literal are treated as significant and appear without changes in the output. This applies to space character, tab character and the line feed. Line feeds or whitespaces within the statement markers are treated as part of the statements and do not appear in the output (since formula language ignores whitespaces, they are harmless).

There is one special case where linefeeds within the literal (and outside markers) are ignored. Normally you embed a statement in the literal using <% and complete the statement using %> in the same line or in another line. There is linefeed between the statement end marker %> and the next line. You normally do not want this line feed to occur in the output.

```
<% foreach(def field in obj.Fields) { %>
:${field.tag}:${field.description}:
<%} %>
```

Line feed after end of stmt

If this line feed is emitted in the output, you would be forced to write the statement and literal part in the same line, as below.

```
<% foreach(def field in obj.Fields){ %>:${field.tag}:${field.description}:<%}%>
```

Since this usage is common, line feed at the end of a line containing a statement is ignored if there is just whitespace between the statement end and the line feed. Similarly line feed preceding a line that contains a statement start is also ignored.

See Also:

Directives
Template String Literal

Directives

Occasionally you may want to control how the template is converted to output (string). For instance, by default, line feeds within the template are emitted as simple line feeds; cartridge return (CR) is never emitted. If you want the output to contain CRLF pairs, you need to direct the template engine accordingly.

Directives are name=value pairs which are treated as instruction to the template engine. They are embedded within the markers `<%@` and `%>`. For instance, the directive `<%@ usecrlf = true %>` instructs the template engine to use CRLF instead of just line feed.

The only directive supported as of now is 'USECRLF'. The value for this directive should either be true or false. If the value is set to true, the template engine will start using CRLF pairs in place of line feeds. You can turn it off later by setting it to false. In the succeeding text (after it is turned off) line feeds will not be converted to CRLF. The default value is false, meaning only line feed is emitted by default.

See Also:

[Expressions](#)
[Scriptlets](#)
[Line Feed and Whitespaces](#)
[Template String Literal](#)

Summary of Template String Syntax

The template string may contain the following,

[Uninterpreted text](#)
[Expressions](#)
[Scriptlets (statements)](#)
[Directives](#)

Of these, the last three appear within special markers. Everything else is treated as uninterpreted text.

Uninterpreted Text

All text that appears within the template string and does not appear within special markers falls under this category. The special markers are used to indicate an expression, scriptlet or a directive. Everything else is treated as uninterpreted text and is passed through as output.

Note that no escape sequences are recognized. This means that uninterpreted text cannot contain '${' and '<%' character sequences, since they will be treated as expression or statement start markers. To workaround this limitation, if your text contains these markers use them as literal within expressions.

```
${"<%"}
```

Expressions

An expression element in a template string is a formula language expression that is evaluated and the result is coerced into a string, which is subsequently emitted into output. The content of an expression must be a complete expression in the formula language.

Two different syntaxes for expressions are supported.

```
<%= expression %>
${expression}
```

Note that you cannot use the %> character sequence as literal characters within an expression using the first syntax. Similarly '}' cannot appear as literal in the second syntax.

Scriptlets (Statement)

Scriptlets can contain any code fragment that is valid for the formula language. Scriptlets need not be fully completed, well formatted statements. When all scriptlets in the template are combined in the order in which they appear, they shall yield a valid statement or sequence thereof. Note that you cannot use the %> character sequence as literal characters within the scriptlet. The suggested workaround is to break it into two literals and concatenate them ("%" + ">").

The syntax of the scriptlet is

```
<% formula script   %>
```

A scriptlet can span multiple lines.

Directives

Directives are instruction to the template engine. The directives have this syntax:

```
<%@ directive   %>
```

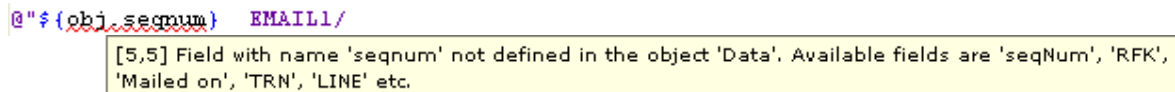There may be optional spaces after the "<%@" and before "%>".

See Also:

Template String Literal

Error Highlighting

The template string literals can contain embedded expressions and statements. Like in normal formula, you are likely to make mistakes while typing it. You may misspell the name of a field; miss a semicolon or a brace etc. It is very important that such errors are caught as early as possible.

In the formula dialog, errors within the literal are highlighted as you type in the same way as normal formula code. This would help you to catch such trivial mistakes as soon as you finish typing.

See Also:

Syntax Highlighting
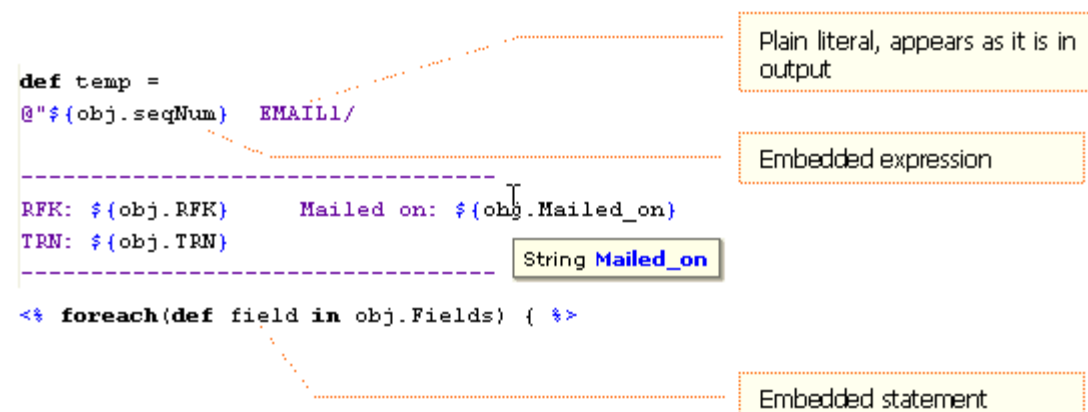Usage Scenario
Comparison with Universal Plug-in
Template String Literal

Syntax Highlighting

The embedded expressions and statements within a template string are syntax highlighted. The literal text, which appears as it is in the output, has the regular literal color. The statements and the expressions are syntax highlighted accordingly. The expression and the statement markers (${} and <% %>) are colored in blue to distinguish it from the literal text.

Also note that, you can locate the definition of the fields/methods used within the literal by pressing the control button down and hovering the mouse over the usage point. A hyper link is shown. Clicking on it will take you to the definition of that variable/function.

See Also:

Error Highlighting
Usage Scenario
Comparison with Universal Plug-in
Template String Literal

Usage Scenario

Template Strings can be used for generating entire messages and it can also be used during mapping to generate a complex field derived from multiple input fields.

See Also:

Error Highlighting
Syntax Highlighting
Comparison with Universal Plug-in
Template String Literal

Simple Strings

It is very common to merge multiple fields or values to form the value of a destination field. These typically mean adding strings using the + operator or the concat function. The output required is broken down to constituent expressions and literals and they are concatenated to get the full output.

```
"/" + Branch_Code + " : " + SeqNum + "//" + FormatDate(Today(),"MMddyyyy")
```

You can use template string literal to achieve the same. For example the above formula code can be changed to literal given below.

```
@"/${Branch_Code} : ${SeqNum}//${FormatDate(Today(),"MMddyyyy")}@"
```

The main advantage is that the literal can easily be correlated with the output generated.

See Also:

Usage Scenario

Complete Messages

If the requirement is to generate semi structured messages dominated by literal text, but also containing values extracted from source, then you should consider using template string to generate it. Since the literal supports multi-line strings and does not escape characters, it is ideal for generating free-form text (like an email). Since messages tend to contain optional and repeating fields, you may have to use embedded 'if' and 'for' statements within the literal.

```
def obj = emailObj.Data;
def HEADER= obj.header[0];
def INPUT= obj.Input[0];
def temp =
@"${obj.seqNum}       EMAIL1/


---------------------------------
RFK: ${obj.RFK}           Mailed on: ${obj.Mailed_on}
TRN: ${obj.TRN}
---------------------------------
* Incoming *
    MT:${HEADER.MTName}
Sender: ${HEADER.Sender}



Send Ref: ${HEADER.Send_Ref}
Receiver:${HEADER.ReceiverRef}
     ${HEADER.ReceiverName}
Owner:${HEADER.Owner}      Internal Priority: ${HEADER.Internal_Priority}
Stage: Inb-Compl          Next Activity: Archive
Input: /F-${INPUT.ltIdentifier}B/${INPUT.SessionNumber}
 /${INPUT.SeqNum}/${INPUT.MessageInputRef}
---------------------------------

<% foreach(def field in obj.Fields) { %>
:${field.tag}:${field.description}:
<% if ( !isNull(field.value)) { %>
${field.value}
<%} %>



<%}  %>


---------------------------------"@;
formulaTempl = temp;
```

Comparison with Universal Plug-in

As mentioned before the string template can be used for generating entire messages, where you have thought of using Universal plug-in to serialize the output. Because these two techniques can be used for solving the same problem, it is very important to understand the benefits and limitations of each solution.

When should you use Universal Plug-in?

String templates can be used only for generating output; it does not help in parsing. While a universal message can be used on both the input and the output side.

Universal messages support both text and binary messages, while string templates can be used only for text. Even in case of text, it allows you specify the format and padding characters for fields. Simulating the same is possible in template string but you have to write the appropriate formula and embed it as an expression.

Universal messages support esoteric possibilities like fixed length fields, length preceded fields, etc. It is difficult to simulate them in template strings. Universal plug-in is ideal for structured messages. For instance, if your message consists of a number of fixed length fields, template string is unlikely to be a good choice.

When should you use Template Strings?

Template Strings are very lightweight. You don't have to define a message structure to use it. You can use it with whatever input field structure you have (for instance an internal message). It is also quite easy to mix data from multiple messages/objects.

Template Strings need not necessarily be used for generating entire messages. It can be used during mapping to generate a complex field derived from multiple input fields.

As the name implies, the string literal is a template from which output is generated. The correlation between the output and template is easy to see. Because of this, it is easy to modify and maintain a template compared to a universal message.

Template strings are ideal for semi-structured messages, like Email. It is ideal for mail merge like scenario, where the output contains lot of predefined text

interspersed with fields that need to be inserted based on some other data source. It can also be considered for generating outputs in HTML, XML or other text formats.

To sum it up, template strings and Universal plug-in cater mostly to different requirements. The use cases where both of them are suitable are minimal. Template strings should only be considered for generating semi structured output, where the output contains lot of predefined text interspersed with fields that need to be inserted based on some other data source. In all other cases use Universal plug-in.

See Also:

Error Highlighting
Syntax Highlighting
Usage Scenario
Template String Literal